Python Dataclasses and Redis Tutorial

Table of Contents

Python Dataclasses

What are Dataclasses?

Dataclasses are a Python feature (introduced in Python 3.7) that automatically generate special methods for classes, reducing boilerplate code. They're perfect for creating classes that primarily store data.

Basic Dataclass Example

```python
from dataclasses import dataclass
from typing import Optional
from datetime import datetime

@dataclass
class User:
    id: int
    name: str
    email: str
    age: Optional[int] = None
    created_at: datetime = None

    def __post_init__(self):
        if self.created_at is None:
            self.created_at = datetime.now()

# Usage
user = User(1, "John Doe", "john@example.com", 25)
print(user)  # User(id=1, name='John Doe', email='john@example.com', age=25, created_at=...)
```

Dataclass Features

```python
from dataclasses import dataclass, field
from typing import List

@dataclass
class Product:
    id: int
    name: str
    price: float
    tags: List[str] = field(default_factory=list)  # Mutable default

    def __post_init__(self):
        # Validation
        if self.price < 0:
            raise ValueError("Price cannot be negative")

@dataclass(frozen=True)  # Immutable
class Point:
    x: float
    y: float

@dataclass(order=True)  # Enables comparison operators
class Score:
    value: int
    player: str = field(compare=False)  # Exclude from comparison
```

When to Use Dataclasses

Use dataclasses when:

- Creating classes that primarily hold data
- You want automatic `__init__`, `__repr__`, `__eq__` methods
- You need type hints and want cleaner syntax
- Working with configuration objects, DTOs, or data containers
- You want immutable objects (with `frozen=True`)

Don't use dataclasses when:

- You need complex inheritance hierarchies
- The class has more behavior than data
- You're working with Python < 3.7

Redis Overview

What is Redis?

Redis is an in-memory data structure store used as a database, cache, and message broker. It supports various data structures like strings, hashes, lists, sets, and more.

Redis Installation and Setup

bash

```bash
# Install Redis server
sudo apt-get install redis-server  # Ubuntu/Debian
brew install redis                 # macOS

# Install Python Redis client
pip install redis
```

Basic Redis Operations

python

```python
import redis
import json

# Connect to Redis
r = redis.Redis(host='localhost', port=6379, db=0)

# String operations
r.set('key', 'value')
value = r.get('key')  # Returns bytes, decode with .decode()

# Hash operations
r.hset('user:1', 'name', 'John')
r.hset('user:1', 'email', 'john@example.com')
user_data = r.hgetall('user:1')

# List operations
r.lpush('tasks', 'task1', 'task2')
tasks = r.lrange('tasks', 0, -1)

# Set operations
r.sadd('tags', 'python', 'redis', 'tutorial')
tags = r.smembers('tags')

# Expiration
r.setex('session:123', 3600, 'session_data')  # Expires in 1 hour
```

When to Use Redis

Use Redis for:

- Caching frequently accessed data
- Session storage
- Real-time analytics and counters
- Message queues and pub/sub
- Temporary data storage
- Rate limiting
- Leaderboards and rankings

Don't use Redis for:

- Primary data storage (use with a persistent database)
- Complex queries (use SQL databases)
- Large objects (Redis is memory-based)
- ACID transactions across multiple operations

Using Dataclasses with Redis

Serialization Helpers

python

```python
import json
import redis
from dataclasses import dataclass, asdict
from typing import Optional, Dict, Any
from datetime import datetime

def serialize_datetime(obj):
    """JSON serializer for datetime objects"""
    if isinstance(obj, datetime):
        return obj.isoformat()
    raise TypeError(f"Object of type {type(obj)} is not JSON serializable")

def deserialize_datetime(date_string: str) -> datetime:
    """Deserialize datetime from ISO format"""
    return datetime.fromisoformat(date_string)

@dataclass
class RedisDataclass:
    """Base class for dataclasses that can be stored in Redis"""

    def to_dict(self) -> Dict[str, Any]:
        return asdict(self)

    def to_json(self) -> str:
        return json.dumps(self.to_dict(), default=serialize_datetime)

    @classmethod
    def from_dict(cls, data: Dict[str, Any]):
        return cls(**data)

    @classmethod
    def from_json(cls, json_str: str):
        data = json.loads(json_str)
        return cls.from_dict(data)
```

Complete Examples

Example 1: User Management System

python

```python
from dataclasses import dataclass
from typing import Optional, List
from datetime import datetime
import redis
import json

@dataclass
class User(RedisDataclass):
    id: int
    username: str
    email: str
    full_name: str
    age: Optional[int] = None
    created_at: Optional[datetime] = None
    last_login: Optional[datetime] = None

    def __post_init__(self):
        if self.created_at is None:
            self.created_at = datetime.now()

class UserRepository:
    def __init__(self, redis_client: redis.Redis):
        self.redis = redis_client
        self.key_prefix = "user:"

    def save(self, user: User) -> bool:
        """Save user to Redis"""
        key = f"{self.key_prefix}{user.id}"
        try:
            self.redis.set(key, user.to_json())
            # Also maintain a set of all user IDs
            self.redis.sadd("user_ids", user.id)
            return True
        except Exception as e:
            print(f"Error saving user: {e}")
            return False

    def get(self, user_id: int) -> Optional[User]:
        """Get user from Redis"""
        key = f"{self.key_prefix}{user_id}"
        data = self.redis.get(key)
        if data:
            return User.from_json(data.decode())
        return None
```

```python
    def get_all(self) -> List[User]:
        """Get all users"""
        user_ids = self.redis.smembers("user_ids")
        users = []
        for user_id in user_ids:
            user = self.get(int(user_id))
            if user:
                users.append(user)
        return users

    def delete(self, user_id: int) -> bool:
        """Delete user from Redis"""
        key = f"{self.key_prefix}{user_id}"
        deleted = self.redis.delete(key)
        self.redis.srem("user_ids", user_id)
        return deleted > 0

    def update_last_login(self, user_id: int):
        """Update user's last login time"""
        user = self.get(user_id)
        if user:
            user.last_login = datetime.now()
            self.save(user)

# Usage
r = redis.Redis(host='localhost', port=6379, db=0)
user_repo = UserRepository(r)

# Create users
user1 = User(1, "johndoe", "john@example.com", "John Doe", 25)
user2 = User(2, "janedoe", "jane@example.com", "Jane Doe", 30)

# Save users
user_repo.save(user1)
user_repo.save(user2)

# Retrieve user
retrieved_user = user_repo.get(1)
print(f"Retrieved: {retrieved_user}")

# Update last login
user_repo.update_last_login(1)

# Get all users
all_users = user_repo.get_all()
print(f"All users: {len(all_users)}")
```

Example 2: E-commerce Product Catalog

python

```python
from dataclasses import dataclass, field
from typing import List, Dict, Optional
from enum import Enum
import redis
import json

class ProductCategory(Enum):
    ELECTRONICS = "electronics"
    CLOTHING = "clothing"
    BOOKS = "books"
    HOME = "home"

@dataclass
class Product(RedisDataclass):
    id: int
    name: str
    description: str
    price: float
    category: ProductCategory
    tags: List[str] = field(default_factory=list)
    stock: int = 0
    rating: float = 0.0
    reviews_count: int = 0

    def __post_init__(self):
        if self.price < 0:
            raise ValueError("Price cannot be negative")

class ProductCatalog:
    def __init__(self, redis_client: redis.Redis):
        self.redis = redis_client
        self.product_key = "product:"
        self.category_key = "category:"
        self.search_key = "search:"

    def add_product(self, product: Product) -> bool:
        """Add product to catalog"""
        try:
            # Save product
            self.redis.set(f"{self.product_key}{product.id}", product.to_json())

            # Add to category index
            self.redis.sadd(f"{self.category_key}{product.category.value}", product.id)

            # Add to search index (by tags)
```

```
            # Add to search index (by tags)
            for tag in product.tags:
                self.redis.sadd(f"{self.search_key}{tag.lower()}", product.id)

            # Update price range for category
            self.redis.zadd(f"price_range:{product.category.value}", {product.id: product.price})

            return True
        except Exception as e:
            print(f"Error adding product: {e}")
            return False

    def get_product(self, product_id: int) -> Optional[Product]:
        """Get product by ID"""
        data = self.redis.get(f"{self.product_key}{product_id}")
        if data:
            product_dict = json.loads(data.decode())
            # Convert category back to enum
            product_dict['category'] = ProductCategory(product_dict['category'])
            return Product.from_dict(product_dict)
        return None

    def get_products_by_category(self, category: ProductCategory) -> List[Product]:
        """Get all products in a category"""
        product_ids = self.redis.smembers(f"{self.category_key}{category.value}")
        products = []
        for product_id in product_ids:
            product = self.get_product(int(product_id))
            if product:
                products.append(product)
        return products

    def search_by_tag(self, tag: str) -> List[Product]:
        """Search products by tag"""
        product_ids = self.redis.smembers(f"{self.search_key}{tag.lower()}")
        products = []
        for product_id in product_ids:
            product = self.get_product(int(product_id))
            if product:
                products.append(product)
        return products

    def get_products_by_price_range(self, category: ProductCategory, min_price: float, max_price: float) -> List[Product]:
        """Get products within price range"""
        product_ids = self.redis.zrangebyscore(
            f"price_range:{category.value}",
            min_price,
```

```
            max_price
        )
        products = []
        for product_id in product_ids:
            product = self.get_product(int(product_id))
            if product:
                products.append(product)
        return products

    def update_stock(self, product_id: int, new_stock: int):
        """Update product stock"""
        product = self.get_product(product_id)
        if product:
            product.stock = new_stock
            self.add_product(product)  # This will update the existing product

# Usage
r = redis.Redis(host='localhost', port=6379, db=0)
catalog = ProductCatalog(r)

# Create products
laptop = Product(
    id=1,
    name="Gaming Laptop",
    description="High-performance gaming laptop",
    price=1299.99,
    category=ProductCategory.ELECTRONICS,
    tags=["gaming", "laptop", "computer"],
    stock=10,
    rating=4.5,
    reviews_count=150
)

book = Product(
    id=2,
    name="Python Programming",
    description="Learn Python programming",
    price=29.99,
    category=ProductCategory.BOOKS,
    tags=["python", "programming", "tutorial"],
    stock=50,
    rating=4.8,
    reviews_count=89
)

# Add to catalog
```

```
catalog.add_product(laptop)
catalog.add_product(book)

# Search and retrieve
electronics = catalog.get_products_by_category(ProductCategory.ELECTRONICS)
python_products = catalog.search_by_tag("python")
affordable_books = catalog.get_products_by_price_range(ProductCategory.BOOKS, 0, 50)

print(f"Electronics: {len(electronics)}")
print(f"Python products: {len(python_products)}")
print(f"Affordable books: {len(affordable_books)}")
```

Example 3: Caching with TTL

python

```python
from dataclasses import dataclass
from typing import Optional
import redis
import time
import json

@dataclass
class CacheItem(RedisDataclass):
    key: str
    value: str
    ttl: int
    created_at: float = None

    def __post_init__(self):
        if self.created_at is None:
            self.created_at = time.time()

class CacheManager:
    def __init__(self, redis_client: redis.Redis):
        self.redis = redis_client

    def set(self, key: str, value: str, ttl: int = 3600):
        """Set cache item with TTL"""
        cache_item = CacheItem(key, value, ttl)
        self.redis.setex(key, ttl, cache_item.to_json())

    def get(self, key: str) -> Optional[str]:
        """Get cache item"""
        data = self.redis.get(key)
        if data:
            cache_item = CacheItem.from_json(data.decode())
            return cache_item.value
        return None

    def get_with_metadata(self, key: str) -> Optional[CacheItem]:
        """Get cache item with metadata"""
        data = self.redis.get(key)
        if data:
            return CacheItem.from_json(data.decode())
        return None

    def delete(self, key: str) -> bool:
        """Delete cache item"""
        return self.redis.delete(key) > 0
```

```
    def exists(self, key: str) -> bool:
        """Check if key exists"""
        return self.redis.exists(key) > 0

    def get_ttl(self, key: str) -> int:
        """Get remaining TTL"""
        return self.redis.ttl(key)

# Usage
r = redis.Redis(host='localhost', port=6379, db=0)
cache = CacheManager(r)

# Cache some data
cache.set("user_profile:123", json.dumps({"name": "John", "age": 30}), ttl=300)
cache.set("api_response:weather", json.dumps({"temp": 25, "humidity": 60}), ttl=600)

# Retrieve data
user_profile = cache.get("user_profile:123")
weather_data = cache.get("api_response:weather")

print(f"User profile: {user_profile}")
print(f"Weather: {weather_data}")
print(f"TTL for user profile: {cache.get_ttl('user_profile:123')} seconds")
```

Best Practices

Dataclasses Best Practices

1. Use type hints for all fields
2. Use `field()` for mutable defaults
3. Implement validation in `__post_init__`
4. Use `frozen=True` for immutable objects
5. Use `order=True` for sortable objects

Redis Best Practices

1. Use connection pooling for production
2. Set appropriate TTL for cached data
3. Use Redis transactions for atomic operations
4. Monitor memory usage
5. Use Redis Cluster for high availability
6. Implement proper error handling
7. Use pipelining for multiple operations

Combined Best Practices

1. Serialize dataclasses to JSON for Redis storage
2. Use consistent key naming conventions
3. Implement proper error handling
4. Use Redis for caching, not primary storage
5. Consider data size when storing in Redis
6. Use Redis pub/sub for real-time updates

This tutorial covers the essential concepts of both dataclasses and Redis, showing how they complement each other in building efficient, maintainable applications.